Preface

This tutorial will go over how objects are placed, loaded, processed, how to edit objects and make new ones. A basic understanding of C code and a working decomp repository is required to understand this tutorial.
Objects are entities that exist in the game world and are updated every frame during normal gameplay. They can be assigned a unique model and behavior, which independently determine how the object acts and how it looks. Almost everything you see that changes state or can be interacted with in game is an object, from rotating platforms and coins, to Mario himself.

Placing objects

Objects are placed by the level script. They can be placed in three different ways: standard objects, macro objects and special objects.
The standard object placement has control over the model id, the position, rotation, behavior parameter (bparam), behavior and available acts of the object; more on what those values do will come later. Special objects are limited to certain model/behavior combos and only take position and yaw input. Macro objects are also limited to certain model/behavior and only take position, yaw and bparam2 inputs.
For the purposes of your average hacker, there is absolutely no reason to ever use anything but a standard object. The other methods were supposed optimization techniques Nintendo used to save space at the cost of limiting inputs, which overall did not save much space and only made editing levels harder. These limitations are way too much for the few bytes saved so I will recommend never using these object placement methods.
All objects, regardless of type, are initialized into the same object pool and share the same struct type, aptly named Object.

The object struct

The object struct contains all the values necessary for object processing. It contains many structs within it that deal with various specific processes, such as graphics, animation and collision. When dealing with objects it helps to separate what values are governed by the object processing system and should not be touched, and which are solely used by specific object behavior's.

```
struct Object
{
    /*0x000*/ struct ObjectNode header;
    /*0x068*/ struct Object *parentObj;
```

```
    /*0x06C*/ struct Object *prevObj;
    /*0x070*/ u32 collidedObjInteractTypes;
    /*0x074*/ s16 activeFlags;
    /*0x076*/ s16 numCollidedObjs;
    /*0x078*/ struct Object *collidedObjs[4];
    /*0x088*/
    union
    {
        // Object fields. See object_fields.h.
        u32 asU32[0x50];
        s32 asS32[0x50];
        s16 asS16[0x50][2];
        f32 asF32[0x50];
        s16 *asS16P[0x50];
        s32 *asS32P[0x50];
        struct Animation **asAnims[0x50];
        struct Waypoint *asWaypoint[0x50];
        struct ChainSegment *asChainSegment[0x50];
        struct Object *asObject[0x50];
        struct Surface *asSurface[0x50];
        void *asVoidPtr[0x50];
        const void *asConstVoidPtr[0x50];
    } rawData;
    /*0x1C8*/ u32 unused1;
    /*0x1CC*/ const BehaviorScript *curBhvCommand;
    /*0x1D0*/ u32 bhvStackIndex;
    /*0x1D4*/ uintptr_t bhvStack[8];
    /*0x1F4*/ s16 bhvDelayTimer;
    /*0x1F6*/ s16 respawnInfoType;
    /*0x1F8*/ f32 hitboxRadius;
    /*0x1FC*/ f32 hitboxHeight;
    /*0x200*/ f32 hurtboxRadius;
    /*0x204*/ f32 hurtboxHeight;
    /*0x208*/ f32 hitboxDownOffset;
    /*0x20C*/ const BehaviorScript *behavior;
    /*0x210*/ u32 unused2;
    /*0x214*/ struct Object *platform;
    /*0x218*/ void *collisionData;
    /*0x21C*/ Mat4 transform;
```

```
    /*0x25C*/ void *respawnInfo;
};
```

Above is the object struct. This and the structs within this can be found inside the types.h file in /include/. Generally you will spend your time inside of the object fields section, which is located inside the union and detailed by the note at offset 0x88. Above that is the graphics section which should not be touched directly as it is usually governed by the object processing engine. Below is data for behaviors and object transformations, which is also used by the object processing engine. You may have to touch these parameters on some occasions, but generally only through helper functions.

Now that you have seen what values you are dealing with when setting up an object, you need to know how behaviors for objects work.

Behavior scripts

Behavior scripts determine how the object acts and appears. Generally the behavior script will initialize the values inside the object struct so that the object is processed properly, as well as tell the game what methods to run on that object.

The behavior script itself is an array where commands are split into multiples of 4 byte chunks and the Most Significant Byte represents the command type. Each cmd type sets specific values inside the object struct, or runs a certain function on that object. Each behavior script starts with setting which object group it will be placed into. Following that are cmds which are only ran once. From there you can either end the script, stopping all behavior processing on the object, or loop the script and get a set of cmds that run every frame. For example, take the following behavior.

```
const BehaviorScript bhvWind[] = {
    BEGIN(OBJ_LIST_UNIMPORTANT),
    OR_INT(oFlags, OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE),
    BEGIN_LOOP(),
        CALL_NATIVE(bhv_wind_loop),
    END_LOOP(),
};
```

It is defined as an array with type BehaviorScript. It begins by setting the object group, and then sets object flags at initialization. Following that is a loop of cmds which run every frame. In this behavior it only runs a function called *bhv_wind_loop*. The most important thing to take note of is that the value oFlags is set on the first frame the behavior is run, and *bhv_wind_loop* is run every frame the behavior is run. Always keep track of what is looped and what is not looped. Also note that cmds that take a struct value (like *oFlags*) are limited to values inside the object fields mentioned above.

There is a lot more to go over on behavior scripts, but rather than cover the cmds I will cover examples of some common script types. For a complete reference of cmds you can find it at the top of the behavior_data.c file inside /data/.

Solid objects

Solid objects, or objects with collision, are things Mario can stand on or collide with based on a defined mesh. Solid objects belong to the object group surface (group 9). Objects not in the surface group with a mesh will have glitchy collision and Mario will not collide with them properly.
Object collision is generated each frame by the game and placed in a dynamic collision pool. This is very inefficient and thus requires object collision to have a small number of triangles or the game will lag a lot (see DDD sub). In order to generate this collision, you must define a collision pointer, and run the function *load_object_collision_model*.
An example of an incredibly simple solid object behavior is as follows:

Simple rotating solid object

```
const BehaviorScript bhvLllRotatingHexagonalPlatform[] = {
    BEGIN(OBJ_LIST_SURFACE),
    OR_INT(oFlags, (OBJ_FLAG_SET_FACE_YAW_TO_MOVE_YAW |
OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE)),
    LOAD_COLLISION_DATA(lll_seg7_collision_hexagonal_platform),
    SET_HOME(),
    BEGIN_LOOP(),
        SET_INT(oAngleVelYaw,  0x100),
        ADD_INT(oMoveAngleYaw, 0x100),
        CALL_NATIVE(load_object_collision_model),
    END_LOOP(),
};
```

Again our behavior is defined with type BehaviorScript and is an array of cmds. This is the case for all behaviors. Remember that behaviors are byte arrays. The type BehaviorScript has length 4 or is equivalent to a u32.
Next note that as a solid platform, we can see it is placed in the solid object group, loads a collision pointer with LOAD_COLLISION_DATA and loads dynamic collision every frame with *load_object_collision_model*. The collision loading ***must*** be done every frame, that is why it ***must*** go inside of the object loop.

Object flags

Another point to take note of is the *oFlags* parameter. This is the most commonly set parameter in a behavior script as it is going to apply the most basic and common

transformations to our objects. The flags names are self explanatory for the most part. You can find a list of all *oFlags* here:

```
/* oFlags */
#define OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE         (1 <<  0) //
0x00000001
#define OBJ_FLAG_MOVE_XZ_USING_FVEL               (1 <<  1) //
0x00000002
#define OBJ_FLAG_MOVE_Y_WITH_TERMINAL_VEL         (1 <<  2) //
0x00000004
#define OBJ_FLAG_SET_FACE_YAW_TO_MOVE_YAW         (1 <<  3) //
0x00000008
#define OBJ_FLAG_SET_FACE_ANGLE_TO_MOVE_ANGLE     (1 <<  4) //
0x00000010
#define OBJ_FLAG_0020                             (1 <<  5) //
0x00000020
#define OBJ_FLAG_COMPUTE_DIST_TO_MARIO            (1 <<  6) //
0x00000040
#define OBJ_FLAG_ACTIVE_FROM_AFAR                 (1 <<  7) //
0x00000080
#define OBJ_FLAG_0100                             (1 <<  8) //
0x00000100
#define OBJ_FLAG_TRANSFORM_RELATIVE_TO_PARENT     (1 <<  9) //
0x00000200
#define OBJ_FLAG_HOLDABLE                         (1 << 10) //
0x00000400
#define OBJ_FLAG_SET_THROW_MATRIX_FROM_TRANSFORM  (1 << 11) //
0x00000800
#define OBJ_FLAG_1000                             (1 << 12) //
0x00001000
#define OBJ_FLAG_COMPUTE_ANGLE_TO_MARIO           (1 << 13) //
0x00002000
#define OBJ_FLAG_PERSISTENT_RESPAWN               (1 << 14) //
0x00004000
#define OBJ_FLAG_8000                             (1 << 15) //
0x00008000
#define OBJ_FLAG_30
```

This list and more are defined inside of *object_constants.h* found in /include/.
So our behavior here has *oFlags* OBJ_FLAG_SET_FACE_YAW_TO_MOVE_YAW and OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE. This means the model will match the

position and rotation of the values inside of the object fields, and that the object's facing yaw will be set to the object's move yaw.
The last part of the behavior is the rotation part. The value of *oAngleVelYaw* is set to 0x100 and *oMoveAngleYaw* is increased by 0x100 every frame. The first sets the yaw velocity for platform displacement, or change in yaw every frame for objects on the platform. Next the move angle yaw is set which is the forward direction of the object. This is not the physical yaw rotation of the object, but the rotation used for determining which direction forward is for physics calculations. This value is copied to *oFaceAngleYaw*, which is the physical rotation, due to our *oFlags* during object processing. Without setting the *oFlags* the way they were earlier this would not actually rotate the object.
So in conclusion this is a solid object added to the solid object group. It gets its *oFlags* and collision initialized on frame one, and on every frame this object is active, its collision is loaded and added to the dynamic collision pool, it gets rotated, and its rotation velocity for platform displacement is set.
This is the simplest type of solid object but also the most useful for creating platforms.

Platform displacement

One thing that may need additional explanation is platform displacement and velocity calculations. Whenever an object is on a platform, and that platform moves, the object on top should also move. The amount those objects move is determined by the object's velocity, and rotational velocity. Though what is strange about the sm64 engine is that angular velocity does not actually update the angle of the platform, so you must separately increase that value, as we see done in the above behavior. For regular speed, you must set the proper oFlag and it will update the position during object processing. Knowing this let's look at some more solid object behaviors.

Solid object with collision distance

```
const BehaviorScript bhvTower[] = {
    BEGIN(OBJ_LIST_SURFACE),
    OR_INT(oFlags, OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE),
    LOAD_COLLISION_DATA(wf_seg7_collision_tower),
    SET_FLOAT(oCollisionDistance, 3000),
    SET_FLOAT(oDrawingDistance, 20000),
    BEGIN_LOOP(),
        CALL_NATIVE(load_object_collision_model),
    END_LOOP(),
};
```

This behavior is the WF tower you see after killing king whomp. Again we see a solid object defined as an array with type BehaviorScript and placed in the solid object group. As usual, *load_object_collision_model* is run every frame to add the object collision to the dynamic collision pool.
What is new here is the SET_FLOAT cmds. The struct values these set are draw and collision distance. These values represent the distance from Mario with which the object will be drawn, and the distance from Mario where collision will be calculated. As you can see, the draw distance is much larger than collision distance.
These values are very important for solid objects. They are initialized to a default value, but that value is usually not adequate for most mesh objects. Draw distance usually should be as large as possible until it is barely visible on the screen. Collision distance should be as small as possible while still encompassing the entire mesh. If not properly set, your object may not act properly if they are too small, or the game may lag if they are too large.

Solid object with callback functions

```
const BehaviorScript bhvFerrisWheelAxle[] = {
    BEGIN(OBJ_LIST_SURFACE),
    OR_INT(oFlags, OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE),
    ADD_INT(oMoveAngleYaw, 0x4000),
    CALL_NATIVE(bhv_ferris_wheel_axle_init),
    BEGIN_LOOP(),
        ADD_INT(oFaceAngleRoll, 400),
        CALL_NATIVE(load_object_collision_model),
    END_LOOP(),
};
```

Here is another rotating platform, this is the ferris wheel platform seen at the beginning of BITDW and the end of BITS. This platform is rotating along the Z axis, otherwise known as roll. We can see the face angle roll is increased by 400 each frame, and the move angle yaw is increased by 0x4000 on frame one. This is equivalent to a quarter turn. This object also does not set a collision pointer, but loads collision every frame. To properly understand this we will have to look at the function that is run, *bhv_ferris_wheel_axle_init*.
Functions for behaviors are all located inside the /src/game/behaviors/ directory. The code files here are all individually separated, usually one file for each behavior but similar behaviors are combined together, like stars. These individual behavior code files are included in aggregate behavior files that usually also contain additional common functions used across several behaviors. These aggregate behavior files are: *behavior_actions.c*, *object_behaviors.c*, and *object_behaviors_2.c*. All the functions

inside those individual behavior files must also be referenced inside of the header file of the appropriate aggregate file.

Finding callback functions

In order to read the function for our behavior we have to open its individual behavior code file. This is troublesome because the file name scheme does not always exactly match the behavior name. I recommend using *git grep -r* to find the file.

```
/mnt/c/hackport/sm64ex-alo$ git grep -r bhv_ferris_wheel_axle_init
data/behavior_data.c:    CALL_NATIVE(bhv_ferris_wheel_axle_init),
src/game/behavior_actions.h:void bhv_ferris_wheel_axle_init(void);
src/game/behaviors/ferris_wheel.inc.c:void
bhv_ferris_wheel_axle_init(void) {
```

Using this method we can search our entire repository for all references of the function. This is useful in general for finding where code is located. From our above search we know our function is inside the file *ferris_wheel.inc.c*.
To recap, our behavior *bhvFerrisWheelAxle*, called the function *bhv_ferris_wheel_axle_init*. This function is located inside an individual code file and that individual file gets included in an aggregate file that concatenates several behavior code files together.

Callback function in depth

If you notice, we pass a function pointer inside the behavior script, this is known as a callback function. This means our function has to follow a standard to work properly. For behavior script code, all callbacks take no arguments and return nothing.
Now we can look at our example function and figure out how it initializes our ferris wheel behavior.

```
void bhv_ferris_wheel_axle_init(void) {
    struct Object *platform;
    s32 i;

    o->collisionData =
segmented_to_virtual(sFerrisWheelProperties[o->oBehParams2ndByte].axl
eCollision);

    for (i = 0; i < 4; i++) {
        platform = spawn_object_relative(i, 0, 0, 0, o,

sFerrisWheelProperties[o->oBehParams2ndByte].platformModel,
```

```
                                            bhvFerrisWheelPlatform);

        if (platform != NULL) {
            platform->collisionData =

segmented_to_virtual(sFerrisWheelProperties[o->oBehParams2ndByte].pla
tformCollision);
        }
    }
}
```

As we can see, we have no arguments and no return. In C this is represented by *void*. As an overview, what this code is doing is setting collision, spawning 4 platforms and then giving them collision. You may also notice use of the variable *o*. This is actually a global variable given a shorthand notation because of how often it is used. The variable it refers to is *gCurrentObject* and that variable is a pointer to the current object being processed. This variable is updated by the object processing loop and is what allows object code to only affect the single instance of the object.
We can see that the collision pointer is set inside the code here.

```
o->collisionData =
segmented_to_virtual(sFerrisWheelProperties[o->oBehParams2ndByte].axl
eCollision);
```

This is necessary because we did not set it inside the behavior script. It is done here because we use different collision depending on the level, while a behavior script is hardcoded to use only one specific collision pointer.

Parent and child objects

Next we have a for loop, and an object spawning function call inside it. Object spawning functions always return a pointer to the spawned object, and also set the value of the parent object variable inside the spawned object's struct.

```
for (i = 0; i < 4; i++) {
        platform = spawn_object_relative(i, 0, 0, 0, o,

sFerrisWheelProperties[o->oBehParams2ndByte].platformModel,
                                            bhvFerrisWheelPlatform);

        if (platform != NULL) {
            platform->collisionData =
```

```
segmented_to_virtual(sFerrisWheelProperties[o->oBehParams2ndByte].pla
tformCollision);
```

To understand the specifics of where the objects go, we need to look at the function *spawn_object_relative*. Functions called for behaviors are almost always inside the object helper files, which of course contain helper code for object code. Our function of interest is as follows.

```
struct Object *spawn_object_relative(s16 behaviorParam, s16
relativePosX, s16 relativePosY, s16 relativePosZ,
                                     struct Object *parent, s32
model, const BehaviorScript *behavior) {
    struct Object *obj = spawn_object_at_origin(parent, 0, model,
behavior);

    obj_copy_pos_and_angle(obj, parent);
    obj_set_parent_relative_pos(obj, relativePosX, relativePosY,
relativePosZ);
    obj_build_relative_transform(obj);

    obj->oBehParams2ndByte = behaviorParam;
    obj->oBehParams = (behaviorParam & 0xFF) << 16;

    return obj;
}
```

There are further depths of functions to look at here to get a complete understanding, but we can infer a lot from just the names and function arguments. The first argument to this function is the bparam, which we set to the value of the iterator of our for loop. The model of the spawned object is set based on the parent's bparam, just like our parent object's collision. Next notice that the spawned object inherits a position and angle from the parent. Then it sets the relative position based on the arguments passed, which for our spawned object is all 0. The build relative transform function simply applies the correct offset based on the parent's angle. So if our object was rotated 90 degrees, it would apply the relative transformation along Z instead of X and so on; though as our transform is 0 that does not matter.

So what we get out of our init function is that it spawns 4 child object platforms with bparams ranging from 0 to 3, all at the same location as our parent. The function also sets the collision and model of our child platform to the proper values, and our parent's collision. Now as a reminder, our object is a solid object that increases its *oMoveAngleYaw* value by 0x4000 and calls this function on initialization. Then

increases its *oFaceAngleRoll* by 400 each frame. To understand how that is used, we must look at the child object behavior.

```
const BehaviorScript bhvFerrisWheelPlatform[] = {
    BEGIN(OBJ_LIST_SURFACE),
    OR_INT(oFlags, OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE),
    BEGIN_LOOP(),
        CALL_NATIVE(bhv_ferris_wheel_platform_update),
        CALL_NATIVE(load_object_collision_model),
    END_LOOP(),
};
```

This is nothing special. Again we must track down the function and read it, which happens to be located in the same file as our parent object's function.

```
void bhv_ferris_wheel_platform_update(void) {
    f32 offsetXZ;
    s16 offsetAngle;

    obj_perform_position_op(POS_OP_SAVE_POSITION);

    offsetAngle = o->parentObj->oFaceAngleRoll + o->oBehParams2ndByte
* 0x4000;
    offsetXZ = 400.0f * coss(offsetAngle);

    o->oPosX = o->parentObj->oPosX + offsetXZ *
sins(o->parentObj->oMoveAngleYaw)
               + 300.0f * coss(o->parentObj->oMoveAngleYaw);

    o->oPosY = o->parentObj->oPosY + 400.0f * sins(offsetAngle);

    o->oPosZ = o->parentObj->oPosZ + offsetXZ *
coss(o->parentObj->oMoveAngleYaw)
               + 300.0f * sins(o->parentObj->oMoveAngleYaw);

    obj_perform_position_op(POS_OP_COMPUTE_VELOCITY);
}
```

So this code is quite heavy on the math, but at a high level we can see that it's setting the position at some offset from the parent's position based on the parent's *oMoveAngleYaw*. The offset angle is 0x4000 times the bparam. 0x4000 is a quarter of a rotation and our bparam ranges from 0 to 3.

Rather than go over the math, what I feel is important here is an overview of how this works. The parent sets its rotation, its collision and spawns the children, setting their bparams and collision. The children look at the parents rotation to know which direction to move, but use their own bparam to decide which position to take. In order to have multiple platforms or have objects interact, you must take care to have a parent and child. One last thing to go over is how this edits position, not velocity. Earlier I wrote that velocity is used to calculate platform displacement, and if this function did not have the position op functions ran, there would be no platform displacement on this applied to Mario. The position op functions convert the change in position to a velocity so platform displacement is properly applied.

Object actions and timers

What is important with solid objects is making sure platform displacement properly gets applied, and that you can do the math to make your object move as you need. We have gone over rotating platforms only so now I want to talk about platforms that change position in cycles.

```
const BehaviorScript bhvSslMovingPyramidWall[] = {
    BEGIN(OBJ_LIST_SURFACE),
    OR_INT(oFlags, (OBJ_FLAG_SET_FACE_ANGLE_TO_MOVE_ANGLE |
OBJ_FLAG_UPDATE_GFX_POS_AND_ANGLE)),
    LOAD_COLLISION_DATA(ssl_seg7_collision_0702808C),
    CALL_NATIVE(bhv_ssl_moving_pyramid_wall_init),
    BEGIN_LOOP(),
        CALL_NATIVE(bhv_ssl_moving_pyramid_wall_loop),
        CALL_NATIVE(load_object_collision_model),
    END_LOOP(),
};
```

This is the behavior of the thin walls inside the SSL pyramid near the top that move up and down. In this behavior we have an initialization and loop function. Let's look at the init first.

```
void bhv_ssl_moving_pyramid_wall_init(void) {
    switch (o->oBehParams2ndByte) {
        case PYRAMID_WALL_BP_POSITION_HIGH:
            break;

        case PYRAMID_WALL_BP_POSITION_MIDDLE:
            o->oPosY -= 256.0f;
            o->oTimer += 50;
            break;
```

```
        case PYRAMID_WALL_BP_POSITION_LOW:
            o->oPosY -= 512.0f;
            o->oAction = PYRAMID_WALL_ACT_MOVING_UP;
            break;
    }
}
```

Here we see three different possible initialization states depending on the bparam. The timer, y position and action can change depending on which state we start in. First let me explain what *oTimer* is. The timer variable, *oTimer*, is one that increases by 1 every processed frame. This operation is done automatically inside the object processing engine. The timer can be reset manually, but also gets auto reset when *oAction* changes. So in conjunction *oAction* and *oTimer* are used to keep track of the internal state of objects. There is also *oSubAction* for when you need multiple actions while keeping the current timer and state. If we look at the loop function we can see how *oAction* and *oTimer* are used.

```
void bhv_ssl_moving_pyramid_wall_loop(void) {
    switch (o->oAction) {
        case PYRAMID_WALL_ACT_MOVING_DOWN:
            o->oVelY = -5.12f;
            if (o->oTimer == 100)
                o->oAction = PYRAMID_WALL_ACT_MOVING_UP;
            break;

        case PYRAMID_WALL_ACT_MOVING_UP:
            o->oVelY = 5.12f;
            if (o->oTimer == 100)
                o->oAction = PYRAMID_WALL_ACT_MOVING_DOWN;
            break;
    }

    o->oPosY += o->oVelY;
}
```

Depending on our action we move in different directions. What changes our action is a certain period of time inside of that action. This creates a cycle where the object is moved up and down repeatedly as it moves through its actions. Also if you return to the init function, you can see the middle state was a midway position. This was achieved by simply taking the middle Y point and timer value and hard setting them on frame one.

We also manually apply the velocity to our position here. This is usually done with an *oFlag* or a function call but the devs decided to manually do it here.
One final point, platform displacement does not get applied as you jump off platforms, so while Mario will move up and down with this platform while standing on it, he will not do so when jumping off which could result in your jump being eaten if the platform moves fast enough.

Conclusion

Objects are placed in levels via the level script. Their actions are dictated by their behavior script, which is made up of an array of byte cmds. These cmds alter values inside the object's struct and allow the object processing engine to properly update the object's state. Most of an object's behavior is defined by callback functions, which are located in separate files and concatenated together using aggregate behavior files.
For solid objects you need to place them inside the solid object group, set a collision pointer and call the load object collision function. When you move the object, you need to make sure it has the proper *oFlags* set so its velocity and position match, and that object's on top of it experience the proper platform displacement as well.
There was a lot to go over here but still much more left to go over. In the next tutorial I will go over interactions and finally there will be one last part where I just go over some examples.